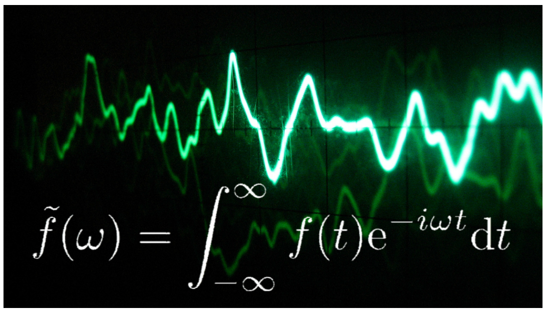
Research Internship (PRE)

Field of Study: SIS/TIC
Scholar Year: 2014-2015

Detecting address sensitivity in Multi Variant Execution Environnment (MVEE)

How to make real life programs compatible with MVEEs


$\tilde{f}(\omega) = \int_{-\infty}^{\infty} f(t)\mathrm{e}^{-i\omega t}\mathrm{d}t$

Confidentiality Notice
Non-confidential report and publishable on Internet

Author:
Thomas Bourguenolle

Promotion:
2016

ENSTA ParisTech Tutor:
Doctor Levy-dit-vehel

Host Organism Tutor:
Doctor Per Larsen

Internship from June 6th 2015 to August 28th 2015

Name of the host organism: Systems Software and Security Lab
Address: University of California, Irvine

Confidentiality Notice

Abstract

Bla bla

Acknowledgment

Bla bla

Contents

Introduction

Bla bla

Part I

Chapter 1

The first chapter is an overview of the history of code injection based attacks and an introduction to Multi Variant Execution Environnments.

I.1 Smashing the stack for decades of research

In 1996, the paper Smashing the stack for fun and profit was released after a few months of buffer overflow attacks. The idea is to exploit the lack of boundary checking while giving the user the opportunity to write a buffer. Thus, the attacker can write over the buffer and overwrite the memory that is over the buffer. A classic way to exploit this is to write shellcode in the upper stack and then change the return address of the function to the beginning of the shellcode. A shellcode is a sequence of commands that result in the opening of a shell, usable by the user which would now have the access rights granted to the original program.

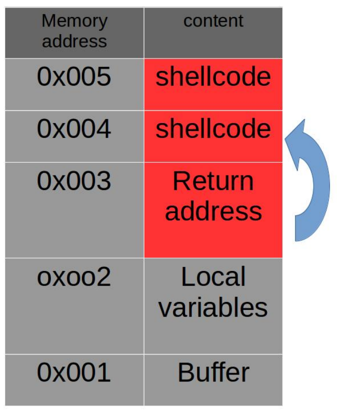


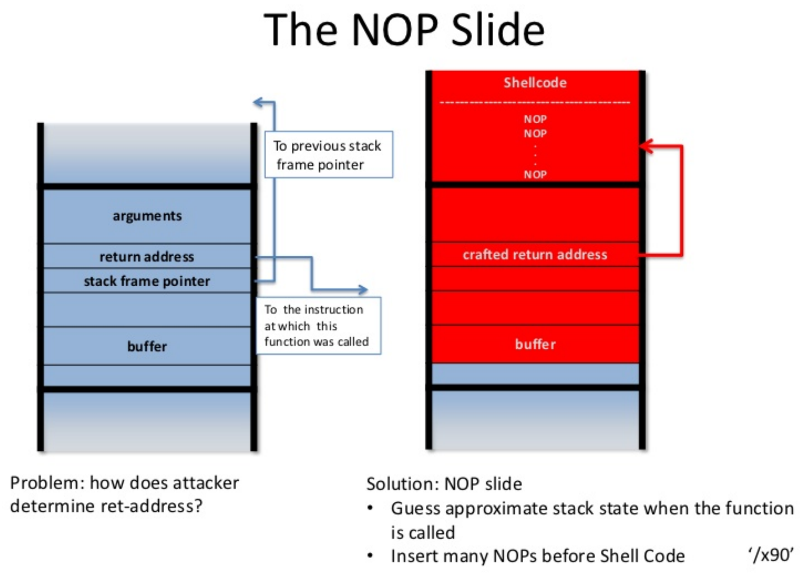
Figure I.1: Memory layout after a buffer overflow attack

To face this problem, many defense mechanisms have been created over the past two decades.

I.2 Address space layout randomization and XOR memory

Address space layout is a simple mechanism that is now enabled by default in any operating system. The idea is just to give the program a different starting memory at every run. As a consequence, the attacker will have a hard time rewritting the return address since he cannot know anymore where his shellcode is located. Nevertheless, the insertion of nop (no operation) instructions before the shellcode allow the attacker to inject a random return address since he could have a high chance (depending of the number of mops) to land on a nop and "slide" to the shellcode.

Another mechanism that is also commonly used is Execute Or Read memory (XOR). In this configuration, every memory address is marked as either readable or executable. As a consequence, shellcode won't be executed since it is very unlikely that all of the shellcode area is marked as executable. Even though this technique seems extremely efficient, attackers have been extremely successful in bypassing this defense and a lot of others.This is the reason why buffer overflows still rank as the third most dangerous attack in the CWE/SANS ranking.



The NOP Slide

Problem: how does attacker determine ret-address?

Solution: NOP slide
- Guess approximate stack state when the function is called
- Insert many NOPs before Shell Code '/x90'

Figure I.2: Nopslide technique

I.3 Return oriented programming

Inserting malicious code being impossible, the new attack model was based on reusing the already available and executable code present in the memory.
The Return into Libc attack, for example, consist in rewritting the return address to the beginning of a known executable memory in the libc library (such as the system function, which allows to spawn a new terminal). One may think that ASLR prevents the attacker from knowing the precise address of this function, but bruteforcing remains efficient on 32bits systems.

Althought, pointer leakage is a way for the attacker to have an exact knowledge of the code layout, allowing him to perform such attacks.
Another very popular attack is ROP (Return Oriented Programming). In this attack, the hacker will use the executable code available by building a gadget. A gadget is simply some chunks of code put together to build a specific set of instruction (spawning a terminal). To do so, the attacker just needs to find instructions chunks ending with the ret instruction. The ret instruction jumps to the addressed referenced by a specific cache. Thus, the attacker will overflow this cache with the consecutive address he needs to jump to (the hacker is hijacking the control flow).

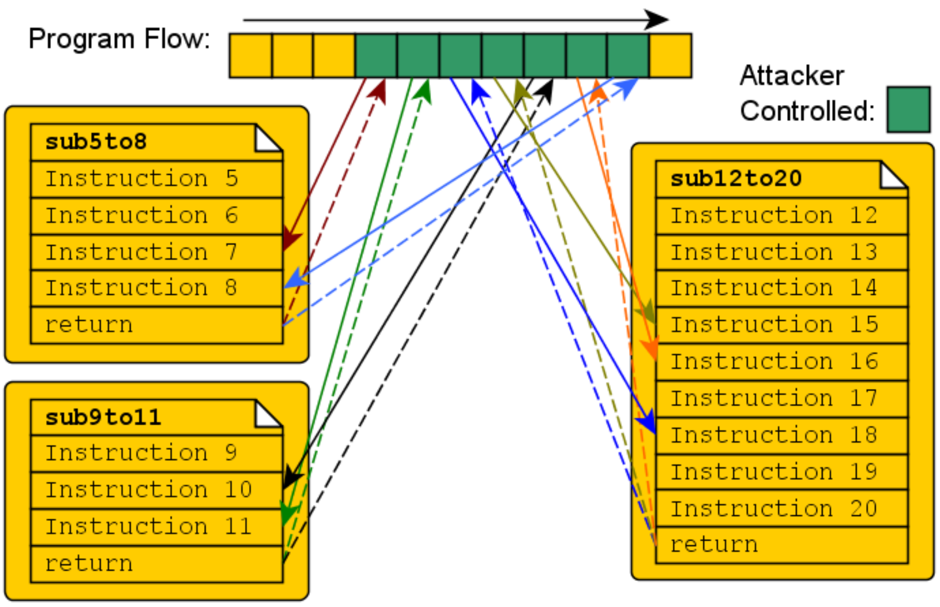


Figure I.3: Building a gadget through ROP

I.4 Multi Variant Execution Environnment

In the MVEE threat model, we assume that the attacker has access to the source code, is able to read all of the memory layout through a leaking pointer and is can successfully develop a gadget (see Return oriented programming). MVEEs consist in running various diversified variants of the same program in parallel. By this mean, if an attacker tries to developp a buffer overflow attack, this attack will only work on one variant. To do so, the MVEE makes sure that the data layout is different in every variant. Then, at runtime, a monitor checks every system call done by the variants, if they are not the same calls or if the arguments differ, the monitor instantly stops every variant and produces a report.

As you can see above, MVEEs induces a consequent time overhead for two reasons: First, the program is run multiple times. And second, variants have to wait for each other when they want to perform a system call (which of course, occurs a lot). However, experience proves that running a MVEE with 2 or 3 variants is still very intersting compared to other defense mechanisms since it is probably one of the most powerful and is still pretty fast compared to

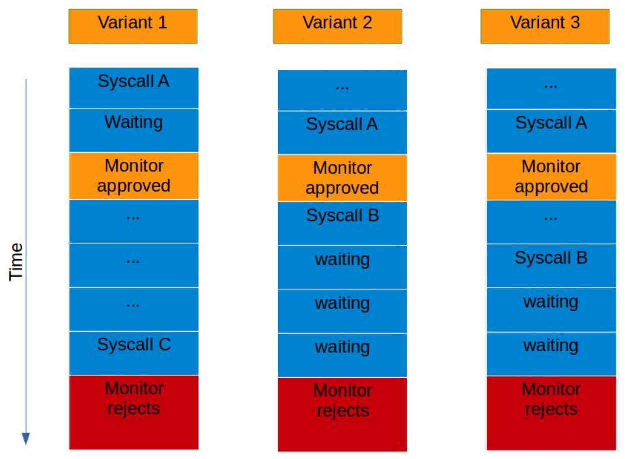


Figure I.4: Monitor the variants inside a MVEE

other techniques such as control flow integrity.
The implementation of a MVEE is a real challenge since a lot of programming issues have to be solved. First the number of system call is important and every single one may require a specific attention. If a pointer is given as an argument, the monitor should make sure that the contents are the same, if an output is created, only one variant should write and we need to prevent the other variants from doing the same operation, etc.
IOCTLs system calls are also fascinating since there are hundreds of them and only a small amount are documented (only 421 of them are documented in the ioctl_list man page and this page informs that this list is very incomplete).

I.5 Exemple d'un tableau

Notation	Description
x	Signal in the time domain
X	Signal in the frequency domain (DFT of x)

Table I.1: Notations

I.6 Exemple pour le glossaire, les acronymes et l'index

Le nom complet : Discrete Fourier Transform (DFT) et juste l'acronyme DFT, ainsi que des ajouts juste dans l'index

I.7 Exemple de citation

Low-Density-Parity-Check (LDPC) [**?**], fountain codes [**?**, **?**], verification codes [**?**] etc.

Conclusion

Bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

List of Tables

List of Figures

Appendix

Bla bla